Concurrent Object-Oriented Programming

Prof. Dr. Bertrand Meyer

Lecture 4: Mutual Exclusion
Material From

The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Overview

• Today we will try to formalize our understanding of mutual exclusion.
• We will also use the opportunity to show you how to argue about and prove various properties in an asynchronous concurrent setting.
• Schedule
  • Formal problem definitions
  • Solutions for 2 threads
  • Solutions for $n$ threads
  • Fair solutions
  • Inherent costs
Mutual Exclusion

- In his 1965 paper E. W. Dijkstra wrote:
  - “Given in this paper is a solution to a problem which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. [...] Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in computer coupling will appreciate the significance of the fact that this problem indeed can be solved.”
Warning

- You will never use these protocols
  - Get over it
- You are advised to understand them
  - The same issues show up everywhere
  - Except hidden and more complex
Time

- “Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external.” (I. Newton, 1689)
- “Time is, like, Nature’s way of making sure that everything doesn’t happen all at once.” (Anonymous, circa 1968)
Events

• An event \( a_0 \) of thread A is
  • Instantaneous
  • No simultaneous events (break ties)
Threads

- A thread $A$ is (formally) a sequence $a_0, a_1, ...$ of events
  - “Trace” model
  - Notation: $a_0 \rightarrow a_1$ indicates order
Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …
Threads are State Machines

Events are transitions
States

- Thread State
  - Program counter
  - Local variables
- System state
  - Object fields (shared variables)
  - Union of thread states
Concurrency

• Thread A
  time

• Thread B
  time
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)
Intervals

- An interval $A_0 = (a_0, a_1)$ is
  - Time between events $a_0$ and $a_1$
Intervals may overlap

\[ \begin{align*}
  & b_0 & B_0 & b_1 \\
  & a_0 & A_0 & a_1 \\
\end{align*} \]

time
Intervals may be disjoint

\[ A_0, a_0, a_1, b_0, b_1, B_0 \]

\[ \text{time} \]
Precedence

- Interval $A_0$ precedes interval $B_0$
Precedence

- Notation: $A_0 \rightarrow B_0$
- Formally
  - End event of $A_0$ before start event of $B_0$
  - Also called “happens before” or “precedes”
Precedence

- Never true that $A \Rightarrow A$
- If $A \Rightarrow B$ then not true that $B \Rightarrow A$
- If $A \Rightarrow B$ & $B \Rightarrow C$ then $A \Rightarrow C$
- Funny thing: $A \Rightarrow B$ & $B \Rightarrow A$ might both be false!
Repeated Events

while (mumble) {
    a_0; a_1;
}

\[ k\text{-th occurrence of event } a_0 \]

\[ a_0^k \]

\[ k\text{-th occurrence of interval } A_0 = (a_0, a_1) \]

\[ A_0^k \]
Implementing a Counter

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
```

Make these steps indivisible using locks
Locks (Mutual Exclusion)

```java
public interface Lock {
    public void lock();
    public void unlock();
}
```

acquire lock
release lock
public class Counter {
    private long value;
    private Lock lock;
    public Long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
Mutual Exclusion

- Let $CS_i^k$ be thread $i$'s $k$-th critical section execution
- And $CS_j^m$ be thread $j$'s $m$-th critical section execution
- Then either $CS_i^k \subseteq CS_j^m$ or $CS_j^m \subseteq CS_i^k$
Deadlock-Free

• If some thread calls lock()
  • And never returns
  • Then other threads must complete lock() and unlock() calls infinitely often
• System as a whole makes progress
  • Even if individuals starve
Starvation-Free

- If some thread calls lock()
  - It will eventually return
- Individual threads make progress
Two-Thread Conventions

class ... implements Lock {
    ...
    // thread-local index, 0 or 1
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        ...
    }
}

Henceforth: i is current thread, j is other thread
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];

    public void lock() {
        // Set my flag
        flag[i] = true;
        // Wait for other flag to go false
        while (flag[j]) {}
    }
}
LockOne Satisfies Mutual Exclusion

- Assume $CS_A^j$ overlaps $CS_B^k$
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction
From the Code

- \( \text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] = \text{false}) \rightarrow CS_A \)
- \( \text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] = \text{false}) \rightarrow CS_B \)

```java
class LockOne implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        while (flag[j]) {}
    }
}
```
From the Assumption

- $read_A(\text{flag}[B]==\text{false}) \rightarrow write_B(\text{flag}[B]==\text{true})$
- $read_B(\text{flag}[A]==\text{false}) \rightarrow write_A(\text{flag}[B]==\text{true})$
Combining

• Assumptions
  • $\text{read}_A(\text{flag}[B]==\text{false}) \Rightarrow \text{write}_B(\text{flag}[B]==\text{true})$
  • $\text{read}_B(\text{flag}[A]==\text{false}) \Rightarrow \text{write}_A(\text{flag}[A]==\text{true})$

• Code
  • $\text{write}_A(\text{flag}[A]==\text{true}) \Rightarrow \text{read}_A(\text{flag}[B]==\text{false})$
  • $\text{write}_B(\text{flag}[B]==\text{true}) \Rightarrow \text{read}_A(\text{flag}[A]==\text{false})$

Impossible
Deadlock Freedom

• LockOne Fails deadlock-freedom
  • Concurrent execution can deadlock
    
    ```
    flag[i] = true;  flag[j] = true;
    while (flag[j]){}  while (flag[i]){}
    ```
  
• Sequential executions OK
public class LockTwo implements Lock {
    private int victim;
    public void lock() {
        victim = i;
        while (victim == i) {}  
    }
    public void unlock() {}  
}
LockTwo Claims

- Satisfies mutual exclusion
  - If thread $i$ in CS
  - Then $\text{victim} = j$
  - Cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```java
public void LockTwo() {
    victim = i;
    while (victim == i) {};
}
```
Peterson’s Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {}
}

public void unlock() {
    flag[i] = false;
}
```
Mutual Exclusion

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {}
}
```

- If thread 0 in critical section
  - flag[0] = true
  - victim = 0

- If thread 1 in critical section
  - flag[1] = true
  - victim = 1

Cannot both be true
public void lock() {
    ...
    while (flag[j] && victim == i) {} 
    ...
}
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {
    }
}

public void unlock() {
    flag[i] = false;
}

• Thread i blocked only if j repeatedly re-enters so that
  • flag[j] == true and victim == j
• When j re-enters
  • it sets victim to j.
  • So i gets in
The Filter Algorithm for n Threads

- There are n-1 “waiting rooms” called levels
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through
class Filter implements Lock {
    int[] level; // level[i] for thread i
    int[] victim; // victim[L] for level L

    public Filter(int n) {
        level = new int[n];
        victim = new int[n];
        for (int i = 1; i < n; i++) {
            level[i] = 0;
        }
    }
    ...
}

Thread 2 at level 4
class Filter implements Lock {
    ...

    public void lock() {
        for (int L = 1; L < n; L++) {
            level[i] = L;
            victim[L] = i;

            while ((∃ k ≠ i : level[k] ≥ L) &&
                   victim[L] == i) {}
        }
    }

    public void unlock() {
        level[i] = 0;
    }
}
Claim

• Start at level $L=0$
• At most $n - L$ threads enter level $L$
• Mutual exclusion at level $L = n - 1$
Induction Hypothesis

- No more than \( n - L + 1 \) at level \( L - 1 \)
- Induction step: by contradiction
- Assume all at level \( L-1 \) enter level \( L \)
- A last to write victim[\( L \)]
- \( B \) is any other thread at level \( L \)
Proof Structure

- Show that A must have seen B in level[L] and since victim[L] == A could not have entered
From the Code and by Assumption

• From the Code
  • (1) $\text{write}_B(\text{level}[B] = L) \Rightarrow \text{write}_B(\text{victim}[L] = B)$
  • (2) $\text{write}_A(\text{victim}[L] = A) \Rightarrow \text{read}_A(\text{level}[B])$

```java
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while ((\exists k \neq i) level[k] \geq L) {
            \&\& \text{victim}[L] == i
        }
    }
}
```

• By Assumption
  • $A$ is the last thread to write $\text{victim}[L]$
  • (3) $\text{write}_B(\text{victim}[L] = B) \Rightarrow \text{write}_A(\text{victim}[L] = A)$
Combining Observations

1. \( \text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A) \)
2. \( \text{read}_A(\text{level}[B]) \)

Thus, \( A \) read \( \text{level}[B] \geq L \). \( A \) was last to write \( \text{victim}[L] \), so it could not have entered level \( L \)!

```java
public void lock() {
    for (int L = 1; L < n; L++) {
        level[i] = L;
        victim[L] = i;
        while ((\exists k \neq i) \text{ level}[k] \geq L) 
            && \text{victim}[L] == i) { }
    }
}
```
No Starvation

• Filter Lock satisfies properties:
  • Just like Peterson Algorithm at any level
  • So no one starves
• But what about fairness?
  • Threads can be overtaken by others
Bounded Waiting

• Want stronger fairness guarantees
• Thread not “overtaken” too much
• Need to adjust definitions ....
Bounded Waiting

- Divide `lock()` method into 2 parts:
  - Doorway interval:
    - Written $D_A$
    - always finishes in finite steps
  - Waiting interval:
    - Written $W_A$
    - may take unbounded steps
r-Bounded Waiting

For threads A and B:
- If $D_A^k \Rightarrow D_B^j$
  - A’s k-th doorway precedes B’s j-th doorway
- Then $CS_A^k \Rightarrow CS_B^{j+r}$
  - A’s k-th critical section precedes B’s (j+r)-th critical section
  - B cannot overtake A by more than r times
- First-come-first-served means $r = 0$. 
Fairness Again

- Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness
    - Not r-bounded for any r!
  - That’s pretty lame...
Bakery Algorithm

- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Lexicographic order
  - \((a,i) > (b,j)\)
    - If \(a > b\), or \(a = b\) and \(i > j\)
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;

    public Bakery(int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false;
            label[i] = 0;
        }
    }
    ...
}

...
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] & & (label[i], i) > (label[k], k)) {}
class Bakery implements Lock {
    ... 
    public void unlock() {
        flag[i] = false;  
    } 
}
No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)
First-Come-First-Served

- If $D_A \rightarrow D_B$ then $A$'s label is smaller
- $\text{write}_A(\text{label}[A]) \rightarrow \text{read}_B(\text{label}[A]) \rightarrow \text{write}_B(\text{label}[B]) \rightarrow \text{read}_B(\text{flag}[A])$
- So $B$ is locked out while $\text{flag}[A]$ is true

```java
class Bakery implements Lock {

public void lock() {
    flag[i] = true;
    label[i] = max(label[0], ..., label[n-1]) + 1;
    while (\exists k \text{ flag}[k]

        && (label[i],i) > (label[k],k)) {}
}
```
Mutual Exclusion

- Suppose A and B in CS together
- Suppose A has earlier label
Mutual Exclusion

- Labels are strictly increasing so B must have seen flag[A] == false
- Labeling_B \rightarrow read_B(flag[A]) \rightarrow write_A(flag[A]) \rightarrow Labeling_A
- Which contradicts the assumption that A has an earlier label
Bakery Y2^32K Bug

```java
class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while (∃k flag[k] && (label[i],i) > (label[k],k));
    }
}
```

Mutex breaks if `label[i]` overflows
timestamps

- Label variable is really a timestamp
- Need ability to
  - Read others’ timestamps
  - Compare them
  - Generate a later timestamp
- Can we do this without overflow?
The Good News

- One can construct a
  - Wait-free (no mutual exclusion)
  - Concurrent
  - Timestamping system
  - That never overflows

This part is hard
Instead...

- We'll construct a sequential timestamping system
  - Same basic idea
  - But simpler
- Uses mutex to read & write atomically
- No good for building locks
  - But useful anyway
Precedence Graphs

- Timestamps form directed graph
- Edge \( x \) to \( y \)
  - Means \( x \) is later timestamp
  - We say \( x \) dominates \( y \)
  - Means \( x \) is later timestamp
  - We say \( x \) dominates \( y \)
Two-Thread Bounded Precedence Graph
Three-Thread Bounded Precedence Graph?

Not clear what to do if one thread gets stuck.
Graph Composition

\[ T^3 = T^2 \times T^2 \]

Replace each vertex with a copy of the graph
Three-Thread Bounded Precedence Graph $T^3$
In General

\[ T^k = T^2 \times T^{k-1} \]

K threads need \(3^k\) nodes
Deep Philosophical Question

- The Bakery Algorithm is
  - Succinct,
  - Elegant, and
  - Fair.
- Q: So why isn’t it practical?
- A: Well, you have to read N distinct variables
Shared Memory

- Shared read/write memory locations called Registers (historical reasons)
- Come in different flavors
  - Multi-Reader-Single-Writer (Flag[ ])
  - Multi-Reader-Multi-Writer (Victim[ ])
  - Not that interesting: SRMW and SRSW
Theorem

- At least $N$ MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.
- $N$ registers like Flag[]...
Proving Algorithmic Impossibility

• To show no algorithm exists:
  • assume by way of contradiction one does,
  • show a bad execution that violates properties:
    • in our case assume an alg for deadlock free mutual exclusion using < N registers
Proof: Need N-MRSW Registers

• Each thread must write to some register.

  \[
  \begin{align*}
    &A \\
    &\downarrow \\
    &CS \\
    &\downarrow \\
    \text{write} \\
  \end{align*}
  \]

  \[
  \begin{align*}
    &B \\
    &\downarrow \\
    &CS \\
    &\downarrow \\
    \text{write} \\
  \end{align*}
  \]

  \[
  \begin{align*}
    &C \\
    &\downarrow \\
    &CS \\
    &\downarrow \\
    \text{write} \\
  \end{align*}
  \]

• ...can’t tell whether A is in critical section
Upper Bound

- Bakery algorithm
  - Uses 2N MRSW registers
- So the bound is (pretty) tight
- But what if we use MRMW registers?
  - Like victim[]?
Bad News Theorem

- At least $N$ MRMW multi-reader/multi-writer registers are needed to solve deadlock-free mutual exclusion.
- Multiple writers don’t help.
Theorem (First 2-Threads)

• Theorem: Deadlock-free mutual exclusion for 2 threads requires at least 2 multi-reader multi-writer registers
• Proof: assume one register suffices and derive a contradiction
Two Thread Execution

- Threads run, reading and writing R
- Deadlock free so at least one gets in
In any protocol B has to write to the register before entering CS, so stop it just before.
Proof: Assume Cover of 1

- A runs, possibly writes to the register, enters CS
- B runs, first obliterating any trace of A, then also enters the critical section
Theorem

• Deadlock-free mutual exclusion for 3 threads requires at least 3 multi-reader multi-writer registers
Proof: Assume Cover of 2

- A writes to one or both registers, enters CS
- Other threads obliterate evidence that A entered CS
- CS looks empty, so another thread gets in
Proof Strategy

- Proved: a contradiction starting from a covering state for 2 registers
- Claim: a covering state for 2 registers is reachable from any state where CS is empty
If we run B through CS 3 times, B must return twice to cover some register, say $R_B$.
Covering State for Two

- Start with B covering register $R_B$ for the 1st time
- Run A until it is about to write to uncovered $R_A$
- Are we done?
- NO! A could have written to $R_B$, so CS no longer looks empty
- Run B obliterating traces of A in $R_B$
- Run B again until it is about to write to $R_B$
- Now we are done
Inductively We Can Show

- There is a covering state
  - Where $k$ threads not in $CS$ cover $k$ distinct registers
  - Proof follows when $k = N - 1$
Summary

• In the 1960’s many incorrect solutions to starvation-free mutual exclusion using RW-registers were published...
• Today we know how to solve FIFO N thread mutual exclusion using 2N RW-Registers
• N RW-Registers inefficient
  • Because writes “cover” older writes
• Need stronger hardware operations
  • that do not have the “covering problem”